

METHOD AND APPARATUS FOR PARTITIONING OF MANAGED STATE FOR A JAVA BASED APPLICATION

(4) Sanjeev Krishnan
19932 Portal Plaza
Cupertino, CA 95014
Citizenship: India

MARTINE PENILLA & KIM, LLP
710 Lakeway Drive, Suite 170
Sunnyvale, CA 94085
Telephone (408) 749-6900

METHOD AND APPARATUS FOR PARTITIONING OF MANAGED STATE FOR A JAVA BASED APPLICATION

by Inventors

Rahul Sharma

Vladimir Matena

Max Mortazavi

Sanjeev Krishnan

CROSS REFERENCE TO RELATED APPLICATIONS

This application is related to (1) U.S. Patent Application No. _____
(Attorney Docket No. SUNMP002A), filed March 19, 2001, and "Method and Apparatus
for Providing Application Specific Strategies to a Java Platform including Start and Stop
Policies," (2) U.S. Patent Application No. _____ (Attorney Docket No.
SUNMP002B), filed March 19, 2001, and entitled "Method and Apparatus for Providing
Application Specific Strategies to a Java Platform including Load Balancing Policies,"
and (3) U.S. Patent Application No. _____ (Attorney Docket No. SUNMP005),
filed March 26, 2001, and entitled "Method and Apparatus for Managing Replicated and
Migration Capable Session State for A Java Platform." Each of these related application
is incorporated herein be reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates generally to Java programming, and more particularly to
methods for managing replicated and migration state in a Java environment.

2. Description of the Related Art

Today's world of computer programming offers many high-level programming languages. Java, for example, has achieved widespread use in a relatively short period of time and is largely attributed with the ubiquitous success of the Internet. The popularity
5 of Java is due, at least in part, to its platform independence, object orientation and dynamic nature. In addition, Java removes many of the tedious and error-prone tasks that must be performed by an application programmer, including memory management and cross-platform porting. In this manner, the Java programmer can better focus on design and functionality issues.

10 One particular Java environment is the Java 2 platform, Enterprise Edition (J2EE), which facilitates building Web-based and enterprise applications. Broadly speaking, J2EE services are performed in the middle tier between the user's browser and the databases and legacy information systems. J2EE comprises a specification, reference implementation and a set of testing suites. J2EE further comprises Enterprise JavaBeans
15 (EJB), JavaServer Pages (JSP), Java servlets, and a plurality of interfaces for linking to information resources in the platform.

The J2EE specifications define how applications should be written for the J2EE environment. Thus the specifications provide the contract between the applications and the J2EE platform. One aspect of the J2EE specification is the EJB 2.0 Container
20 Managed Persistence (CMP). The EJB 2.0 specification defines a contract between an entity bean, its container and the persistence manager for the management of persistent state and relationships for the entity beans. For a complete specification of CMP, refer to

the EJB 2.0 specification published by Sun Microsystems, Inc., which is incorporated by reference herein in its entirety.

According to the EJB programming model, a bean provider develops a set of entity beans for an application and specifies the relationships between these objects. For each entity bean, the bean provider specifies an abstract persistence schema, which defines a set of methods for accessing the container-managed fields and relationships for the entity bean. . The container-managed fields and relationships of the abstract persistence schema are specified in the deployment descriptor defined by the bean provider.

The deployer uses the persistence manager provider tools to determine how persistent fields and relationships are mapped to the underlying persistence mechanism, such as, a database. The persistence manager tools also generate the additional classes and interfaces that enable the persistence manager to manage the persistent fields and relationships of the entity beans at the runtime. An advantage of container managed persistence is that the entity beans become logically independent of the underlying persistence mechanism. The CMP also leads to a simple programming mode for managing persistence.

An entity bean with container manager persistence includes its class, a remote or local interface that defines its client-view business methods, a home interface that defines create, remove, home and finder methods. The abstract persistence schema comprises a set of properties, each representing a field or relationship in the persistent state of entity bean. The entity bean defines a set of accessor (setter and getters) methods for the persistent fields and relationships.

The bean provider generally does not write any database access calls in the entity bean class. Instead, a persistence manager that is available to the container at runtime handles the persistence. The bean provider codes all persistent data access using the setter and getter methods defined for the container-managed persistent and relationship fields.

Figure 1 is a diagram showing the tradeoffs that business and carrier grade applications have to make. The carrier grade applications, which are applications used in high-performance, high-traffic networks such as used by telecoms, service providers and ISPs, require higher performance and availability than business applications.

Carrier grade (CG) applications require high availability of the order of five 9's and better. A high availability environment is one in which a service or component has greater availability, usually due to component redundancy, than in some base environment. Typically the term is used to describe failover cluster environment in which a service is provided by a primary component, until after some failure, after which the secondary component takes over the provision of the service. The high availability requirement for carrier grade applications leads to a requirement for a carrier grade application to achieve prompt (two seconds or less) restart or failover to a secondary component with minimal disruption of service. Thus, the application has to become operational within minimal time after a failure.

A carrier grade application requires a shorter failover time as compared to the business applications, which typically store persistent state in a database to achieve ACID (Atomicity, Consistency, Isolation, Durability) properties for the managed state. Business

applications rely on database-specific mechanisms to achieve state replication, thereby protecting persistent data from the failure of the primary database.

A typical database replication mechanism is database log replay. The database logs changes in a transaction log that is used for transaction replay in case of a failure.

- 5 The log replay involves applying the transaction log to a replica database so that a near-mirror copy of primary database is created. Unfortunately, the time-delay in replaying the transaction log on the replica database slows down the failover. Moreover, a huge rate of inserts/updates may create huge transaction replay log, which further slows the transaction replay at the failure time.

- 10 Business applications also rely on parallel database servers to achieve state replication and failover. In case of parallel database servers, multiple active database engines, coordinated by a distributed lock manager, manage active replication. However, use of the distributed lock manager for coordinating database operations slows down the application performance. Hence, the use of a parallel database server is considered more
- 15 suitable for read-only and read-mostly applications. With the short failover time requirement, a typical carrier grade application cannot rely on a database-based replication mechanism or a parallel database server to achieve state replication and failover.

- Business applications typically need to maintain ACID properties for the data
- 20 being used by the application. Such applications cannot afford any data inconsistency and thereby store persistent data in a database and use transactions. For business applications, the tradeoff between consistency and concurrency gets reflected in the choice of the database isolation level. The use of optimistic concurrency model as against pessimistic

concurrency model is another design decision involved in the business applications as part of the consistency and concurrency tradeoff. Business applications also need a reliable and consistent database failover and recovery—these applications cannot afford any inconsistent data.

5 The carrier grade applications may have consistency and concurrency requirements that differ from the business applications. A carrier grade application may process multiple concurrent state transitions and may not require full ACID properties to be maintained for its managed state. For example, a CG J2EE application may need a fast failover and may afford to have the replica (now the new primary) take over in a state
10 that is temporally inconsistent with the initial primary. The client using the carrier grade application should be able to bring the new primary into a consistent state by retrying transactions for state transitions.

 In view of the forgoing, there is a need for systems and methods that manage state for applications that require high availability. The systems and methods should allow for
15 state replication and provide high availability of the order of five 9's and better to carrier grade applications.

SUMMARY OF THE INVENTION

Broadly speaking, the present invention fills these needs by providing state management units that allow application developers to specify the mechanism and policies of replicated state management to be used with the entity beans of the application. In one embodiment a method is disclosed for partitioning managed session state for a Java based application comprising of entity beans. The application developer is allowed to classify individual entity beans with a particular state management type. Then, during execution, a plurality of state objects are provided, where each state object stores the state of a corresponding entity bean object within the memory address space of a Java server process.

Further, each state object is associated with the state management type of the corresponding entity bean object. The state management type identifies the mechanism and policy for replication of state objects to the different types of state servers and for migration of the state objects from one server process to another. In this manner, state management is provided for each entity bean object based on the associated state management type using a corresponding state object. The state management type can be a memory replicated state management type, a disk replicated state management type, a non-replicated state manage type, or other state management type that may be needed for a entity bean. To facilitate state management the state objects are grouped based on the type of state management associated with the corresponding entity bean object.

A further method for partitioning container-managed state for a Java application is disclosed in another embodiment of the present invention. The method includes partitioning individual entity bean objects of the Java application into state partitions,

wherein the state partitions manage concurrency for the Java application. In addition, individual entity bean objects are classified within each state partition using state management units, wherein each state management unit identifies a particular state management mechanism for recoverable and migration capable state of corresponding entity bean objects. Generally, the state management units provide a mechanism to checkpoint the state objects to state servers, where a replica of each state management unit is maintained. A control module of the application can be used to manage dynamic partitioning of the state of the application via the state partitions and the state management units. In addition, the state partitions and state management units are modular such that additional state management types for the state management units can be defined. Generally, to manage concurrency, each state partition serializes transactions for entity bean objects within a particular state partition by allowing only one concurrent transaction to be performed on the entity bean objects within the particular state partition during a given time period.

In another embodiment, a system for partitioning managed session state for a Java based application is disclosed. The system includes an application having a plurality of entity bean objects. Also included in the system is a plurality of state objects, each state object storing a state of a corresponding entity bean object within a memory address space of a Java server process. In addition, each state object is associated with a particular state management type. State management units are used classify the state objects based on the particular state management type associated with each state object. The state management units facilitate state management for each entity bean object. The state management type can be a memory replicated state management type, or a disk replicated state management type. Further, the state management units can be used to

manage checkpoints using the state objects, and also can be used to perform lock management using the state objects.

Advantageously, the embodiments of the present invention manage the state of entity beans using an in-memory state management facility, which allows state replication
5 both in memory and on disk. As a result, applications executing on the Java system of the embodiments of the present invention can, in some embodiments, achieve continuous availability, on the order of about 99.9999% uptime or better. Other aspects and advantages of the invention will become apparent from the following detailed description, taken in conjunction with the accompanying drawings, illustrating by way of example the
10 principles of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention, together with further advantages thereof, may best be understood by reference to the following description taken in conjunction with the accompanying drawings in which:

5 Figure 1 is a diagram showing the tradeoffs that business and carrier grade applications have to make;

Figure 2 is a Java system having state management, in accordance with an embodiment of the present invention;

Figure 3 is a block diagram showing replicated state subsystems, in accordance
10 with an embodiment of the present invention;

Figure 4 is a diagram showing state management types for state management units (SMU), in accordance with an embodiment of the present invention;

Figure 5 is an illustration showing a hierarchy of use operations for an RSM, in accordance with an embodiment of the present invention;

15 Figure 6 is a class diagram showing repository interfaces used by the RSM, in accordance with an embodiment of the present invention;

Figure 7 is a sequence diagram showing an initialize EJB module state sequence, in accordance with an embodiment of the present invention;

Figure 8 is a sequence diagram showing a manage state sequence, in accordance
20 with an embodiment of the present invention;

Figure 9 is a sequence diagram showing a checkpoint managing sequence, in accordance with an embodiment of the present invention; and

Figure 10 is a sequence diagram showing a lock managing sequence, in accordance with an embodiment of the present invention.

5

www.sun.com

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

An invention is disclosed for systems and methods that provide partitioning of managed state for Java based applications, including applications requiring high availability. To this end, embodiments of the present invention provide a subsystem that manages the replicated and migration capable state for an Enterprise Java Bean (EJB) application using state management units. The state management units allow application developers to partition the state of a Java application into various state management types, allowing the developer to fine tune state management for an application to suit requirements for the application. For example, depending on the application requirements, an application developer or architect may choose to partition the state of an application such that part of the state is replicated to a memory-based state server for better transaction throughput than the part of the state that is replicated to a disk-based state server. In addition, part of the application state can be made non-replicated provided this state does not need to be capable of recovery after an application failure.

In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention may be practiced without some or all of these specific details. In other instances, well known process steps have not been described in detail in order not to unnecessarily obscure the present invention.

Figure 1 has been described in terms of the prior art. Figure 2 is a Java system having state management, in accordance with an embodiment of the present invention. The Java system 200 includes an application runtime subsystem 202 having a

Java 2 Enterprise Edition (J2EE) Server 204 and a replicated state manager (RSM) subsystem 206. Also included in the Java system 200 are a Java application 208, a repository subsystem 210, a disk replicated state server subsystem 212, and a memory replicated state server subsystem 214.

5 Figure 2 shows the RSM 206 interfacing with the (J2EE) Server 204, the Java application 208, the repository 210, the disk replicated state server subsystem 212, and the memory replicated state server subsystem 214. The J2EE Server subsystem 204 provides the runtime environment for J2EE applications, and includes implementations of different types of J2EE containers, such as an application client container, an EJB
10 container, and a web container.

 The Repository subsystem 210 stores and manages the software load for the J2EE application 208 and J2EE server 204. During pre-deployment of a J2EE application 208, classes and a deployment descriptor provided by an application developer are loaded into the repository 210, and later, loaded on to a J2EE server 204 to make the application 208
15 operational.

 State server subsystems store and manage the replicated state for the J2EE application 208, which makes an application highly available. During application runtime, the RSM 206 replicates the replicated state to a state server. If the application 208 running on the J2EE server 204 fails, the application 208 restarts after failure by
20 recovering its state from the replicated state server. The Java system 200 provides two types of state servers, a memory replicated state server 214 and a disk replicated state server 212. The memory replicated state server 214 stores the replicated state in an in-

memory database, while disk replicated state server 212 uses a disk-based database to store and manage the replicated state.

As shown in Figure 2, the RSM subsystem 206 is part of the J2EE Application Runtime subsystem 202, which is responsible for management and supervision of running the J2EE application 208. The RSM 206 manages the replicated and migration capable state for J2EE applications 208 that are running on a J2EE server 204. By managing application state, the RSM 204 provides support for online application upgrades, failure recovery and load balancing features of the J2EE system 200. The RSM 206 uses a memory database within a J2EE server process 204 to manage the application state. In this manner, the RSM enables an application to remain operational even if the state servers become temporary unavailable.

Figure 3 is a block diagram showing replicated state subsystems 300, in accordance with an embodiment of the present invention. The replicated state subsystems 300 show a J2EE server 204 in communication with a disk replicated state server 212, and a memory replicated state server 214. Based on the CMP model, the application developer develops a set of entity beans 304 for an application and specifies the relationships between these objects. For each entity bean 304, the application developer specifies an abstract schema that defines a set of methods for accessing the container-managed fields and relationships for the entity bean. These container-managed fields and relationships are specified in the deployment descriptor defined by the application developer.

Each entity bean 304 includes an abstract class, a local and/or remote interface that defines bean's client-view business methods, a home interface that defines create,

remove, home and finder methods. The abstract schema includes a set of properties, with each property representing a field or relationship in the container-managed state of the entity bean 304. The entity bean 304 also defines a set of setter and getters methods for the container-managed fields and relationships. The application developer codes state
5 access and modifications using the setter and getter methods defined for the container-managed fields and relationships.

The EJB 2.0 CMP specification sets forth, without specific implementations, goals for a persistence manager that provides management of persistent state and relationships for the entity beans 304. The embodiments of the present invention provide
10 specific methods for state management that achieve the goals set forth by the EJB 2.0 CMP specification. Specifically, the RSM of the embodiments of the present invention is a carrier-grade J2EE specific implementation of a persistence manager facility.

However, instead of mapping an abstract schema to a database-based persistence mechanism, RSM manages the state of entity beans 304 and dependent objects 306 using
15 an in-memory state management facility. This in-memory state manager runs within a J2EE server process. To support state recovery during an application restart and migration, the RSM actively replicates the in-memory state to disk replicated state servers 212 and/or memory replicated state servers 214.

Figure 3 shows two EJB modules 302 deployed on a J2EE server process 204.
20 Each EJB module 302 includes a set of entity beans 304. The RSM defines a separation between the application part 350 and managed state part 352.

The application part 350 includes abstract entity bean object classes 304 provided by application developer, based on the EJB 2.0 CMP model. The application part 350 also includes concrete implementation classes generated by the RSM. Further, the application part 350 provides the implementation of methods that provide the state transition logic for a J2EE application.

The Managed state part 352 includes the state objects 314 that capture the state of entity beans 304. The separation of state between application part 350 and managed state part 352 enables the RSM to support both application upgrade and migration.

The RSM of the embodiments of the present invention can generate concrete implementation classes for entity beans 304. The concrete classes generated by the RSM are responsible for managing the recoverable state of the entity beans 304. The RSM also provides implementation of collection classes that are used in managing container-managed relationships. By providing implementation of the getter and setter methods of the corresponding abstract classes, the RSM can implement the entity bean 304 classes. The RSM can also manage the mapping between primary keys and EJB objects, and can store the recoverable references to the remote and home interfaces of other EJBs.

The RSM further manages the relationships between entity beans 304. This includes maintaining the referential integrity of the container-managed relationships in accordance with the semantics of the relationship type.

The RSM manages the recoverable state of entity bean 304 based on the type of the state. Depending on the type of state, the RSM replicates the state in either a disk replicated state server 212 or a memory replicated state server 214. In addition, the RSM

makes application state capable of migration from one J2EE server process 204 to another J2EE server process.

The RSM supports checkpoints of the recoverable state to the two types of state servers, namely, the disk replicated state server 212 and the memory replicated state server 214. This includes support for connecting to the servers, sending checkpoints, recovering replicated state and merging the recovered state into existing in-memory state. Further, the RSM recovers the replicated state from the state servers during application restart after a failure or shutdown or during migration of EJB module from one J2EE server process to another.

During pre-deployment of an EJB module 302, the RSM maps the abstract schema of entity beans 304 classes to a physical schema used by the RSM. To perform this form of schema mapping, the RSM can use a deployment descriptor of EJB components. The RSM generates concrete implementations for the entity bean 304 classes defined as abstract classes by the application developer. A concrete implementation class includes the code that implements the setter and getter methods for container-managed fields and relationships based on the RSM mechanism.

The embodiments of the present application allow state managed by the RSM for a J2EE application to be partitioned into multiple state partitions. In some embodiments of the present invention a state partition defines a unit of concurrency. In these embodiments, the RSM serializes access to a state partition from concurrent transactions. For example, using this embodiment, if a J2EE application includes multiple entity beans having states managed as part of a single state partition, then only a single transaction can be active across the entity bean instances in this state partition at any particular instance.

During application design, application designer partitions the replicated and migration capable state of an application using State Management Units of different types and State Partition. Such state partitioning may be specified using an application configuration descriptor or done dynamically through control module. The repository
5 maintains a representation for the static partitioning of state using schema archives. Schema archives are described in greater detail in U.S. Patent Application No. _____ (Attorney Docket No. SUNMP005), filed March 26, 2001, and entitled "Method and Apparatus for Managing Replicated and Migration Capable Session State for A Java Platform," which is incorporated herein be reference in its entirety.

10 RSM partitions entity beans based on the range of primary keys that will be served by different state partitions. From example, account 1 to 100 will be managed within a single state partition 1, while account 101 to 200 will be managed in a separate state partition 2. A state partition also identifies the unit of concurrency. J2EE server process and RSM serialize transactional access to a state partition. At any specific instance, there
15 can be only one transaction active on entity beans within a state partition.

After partitioning state in to State Partitions, application designer partitions each state partition in to multiple State Management Units. A state partition can contain multiple state management units (SMU) of different types.

Figure 4 is a diagram showing state management types 400 for state management
20 units (SMU), in accordance with an embodiment of the present invention. The state management types 400 are divided into a non-recoverable state 402 and recoverable states 404. The non-recoverable state 402 includes a non-replicated state 406. The recoverable states include a disk replicated state 408 and a memory replicated state 410.

A SMU is a collection of state objects 314 with the same state management type 400, and further defines a unit of checkpoints for the recoverable state types 404. An application can include multiple SMUs of different types depending on the specific requirements of the application. The RSM replicates a disk replicated SMU 408 to a disk replicated state server. The RSM is then capable of automatically recovering the disk replicated SMU 408 during an application restart after failure, shutdown or migration.

The RSM replicates a memory replicated SMU 410 to a memory replicated state server, and is then capable of automatically recovering memory replicated SMU 410 during an application restart after failure, shutdown or migration. The not replicated SMU 406 is not replicated by the RSM to either a disk replicated state server or a memory replicated state server. Generally, the RSM manages a non-replicated SMU 406 to support the migration state of a J2EE application from one J2EE server process to another.

Figure 5 is an illustration showing a hierarchy of use cases 500 for an RSM, in accordance with an embodiment of the present invention. During application runtime, a Control Module 550, an EJB Container 552, a Transaction Manager 556, and an EJB Client 554 are the actors that drive use cases for the RSM.

The control module 550 is a part of a Java application that provides control and application-specific policies for the application. The control module 550 is described in greater detail in related U.S. Patent Application No. _____ (Attorney Docket No. SUNMP002A), filed March 19, 2001, and entitled "Method and Apparatus for Providing Application Specific Strategies to a Java Platform including Start and Stop Policies," which is incorporated by reference in its entirety. The control module 550 is responsible

for supervising the J2EE server 204 and the EJB modules at application runtime. Since the RSM is part of application runtime for a J2EE server process 204, J2EE server 204 activates the RSM as part of the start J2EE server process 502, the upgrade module process 504, and the move module process 506. The control module 550 interacts with
5 the J2EE server 204, which in turn drives uses cases for RSM.

The EJB client 554 invokes a method on an entity bean that has its state managed by the RSM using a manage application state process 510. Such method invocation generally happens under a transaction, and drives the RSM to manage any changes to the entity bean's state as part of the invocation. The EJB container 552, which is part of
10 J2EE server 204, intercepts the method invocation to inject its services, for example, the container 552 can start a transaction that brackets a method invocation. After injecting its services, the container 552 dispatches method invocation to the target entity bean instance. The EJB container 552 interfaces with the RSM to drive the manage transactions use case 512, the manage application state use case 510, and the manage
15 checkpoints use case 514. Finally, the transaction Manager 556 manages transactions for the RSM, which acts as a transactional resource manager.

The RSM uses the manage application state use case 510 to manage state for an EJB-based application. The RSM acts as a carrier-grade implementation of the persistence manager facility defined in the EJB 2.0 specification. CMP model uses the
20 RSM to manage replicated and migration-capable state for entity bean.

Initially, the EJB client 554 makes an invocation on a method defined as part of the remote or local interface of the target entity bean. Typically, a method invocation maps to a state transition implemented by the entity bean. Based on the CG J2EE

programming model, a method invocation on an entity bean can be either local or distributed, synchronous or asynchronous, but preferably does not have any affect on how the RSM manages state for an invoked entity bean instance.

The EJB container 552 then intercepts the method invocation from the EJB client.

5 The EJB container 552 uses this interception to inject container-specific services, such as, transaction bracketing and security mapping. As part of the method implementation, abstract entity bean classes provided by the application developer invoke setter and getter methods for container-managed fields and relationships. A concrete implementation class is then generated by the RSM and implements the setter and getter methods. This
10 enables the RSM to manage the state of container-managed fields and relationships as part of its implementation. The RSM manages the state of entity beans using an in-memory state management facility, which runs within a J2EE server process.

When an entity bean undergoes state transitions initiated by a client application, the application changes the state objects. Any changes to the managed state part are
15 tracked by the RSM. Depending on the state management requirements specified for a J2EE application, the RSM replicates the state objects to disk replicated and memory replicated state servers.

The RSM uses the manage checkpoint process 514 to manage checkpoints of replicated state, and then issues checkpoints to the state servers. More specifically, the
20 RSM implements a checkpoint mechanism that is configurable using a checkpoint policy, which can be specified by the application Control Module 550. The RSM then uses the checkpoint mechanism to replicate state to the disk replicated state server and the memory replicated state server.

5 The RSM can issue checkpoints at different points, such as at the successful commit of each transaction. Generally, no checkpoint is issued for a transaction that fails to commit and is rolled back. This ensures that the state is replicated to state servers for only committed transactions leading to consistent recovery after failure, migration or shutdown. The RSM can optimize the checkpoint mechanism by combining transactions from successful commits of multiple transactions, and maintaining a sequence of box-carried checkpoints. This is referred to as boxcarring of checkpoints for multiple transactions.

10 Further, the checkpoint can be issued either synchronously or asynchronously. The synchronous checkpointing increases the reliability associated with the checkpointing mechanism. The state server ensures that checkpoints propagated to it are processed, thereby avoiding any potential loss of checkpoints. The replicated state in the state servers stays identical with the in memory state managed by RSM, ensuring a faster recovery during the failover.

15 In the asynchronous checkpoint mechanism, the RSM on the J2EE server process can enqueue checkpoints on a local message queue, thus the checkpoint messages are placed in the local address space before getting dispatched. The checkpoint operation returns immediately after the enqueueing operation allowing the on-going transaction to complete.

20 The message queue takes the responsibility of delivering the checkpoint messages to the state server at some time later after the transaction has been successfully committed. Preferably, the message queue preserves the ordering of the checkpoint messages in the order that transactions were executed on the application.

Asynchronous checkpointing adds more flexibility to the state replication mechanism. Based on the checkpointing policy, the message queue can take decision to propagate asynchronous checkpoints at different intervals—after each committed transaction, after a set of committed transactions or after a defined time interval.

5 Figure 6 is a class diagram showing repository interfaces 600 used by the RSM, in accordance with an embodiment of the present invention. The repository maintains classes and configuration descriptors that represent partitioning of state for a J2EE application. The repository interfaces 600 includes EJB modules 302, a logical schema archives 602, a physical scheme archives 604, entity bean classes 606, SMU archives
10 610, and state object classes 612.

The RSM maintains both a logical schema archive 602 and a physical schema archive 604 for each EJB module 302, which are populated in the repository 210 during pre-deployment of an EJB module 302. More specifically, during pre-deployment of the EJB module 302, the RSM creates a logical schema archive 602 in the repository, which
15 includes abstract classes for entity beans 606. The logical schema archive 602 also includes concrete implementation classes that are generated by the RSM. The RSM maps the logical schema archive of a pre-deployed J2EE application to the application part at runtime.

The physical schema archive 604 includes the state object classes 612 and RSM-
20 generated artifacts used for the managed state part of a J2EE application. The SMU archive 610 groups state object classes 612 based on the state management type. Multiple SMU archives 610 can be included within a physical schema archive 604. Both logical

and physical schema archives are loaded in the repository as part of application software load.

Figure 7 is a sequence diagram showing an initialize EJB module state sequence 700, in accordance with an embodiment of the present invention. The EJB module state sequence 700 illustrates how an RSM for an EJB module is started. In operation 701 and 702, the control module 750 for an application starts up an EJB module 302 on a J2EE server process 204. An EJB module 302 may need to be initialized because the EJB module 302 is being started on a J2EE server process 204 after application failure or shutdown, or because the EJB module 302 has been migrated from one J2EE server process 204 to another. The EJB module 302 then requests to the RSM 206 to load the physical schema archive for the EJB module 302, in operation 703.

In operation 704, the RSM 206 loads the physical schema archive from the repository 210, and in operation 705 the RSM 206 loads the schema archive using the RSM class loader 752. In operations 707-713 the RSM creates state partitions 752 and SMUs 754 based on the specification provided by the application control module 750. State Partitions and SMUs can be either created statically using partition configuration descriptor loaded from the repository or managed dynamically by the control module 750. This diagram shows the creation of partitions and SMU based on the specification provided by the application control module 750. Then, in operations 714 and 715 the RSM 206 recovers any replicated state for this EJB module and its set of SMUs from the two state servers.

Figure 8 is a sequence diagram showing manage state sequence 800, in accordance with an embodiment of the present invention. The manage state sequence

800 illustrates a well-defined contract between an RSM_OrderBean 852, which is a concrete implementation class, and an OrderState 854, which is a subtype of state object. The contract illustrated by the manage state sequence 800 is defined by an EJB_RSM Contract interface that is implemented by the SMU 754, which includes the state objects.

5 In operation 801, an EJB client 554 makes an invocation on a method defined as part of the remote interface (or local interface) of the target entity bean. Typically, a remote method maps to a state transition implemented by the entity bean. As mentioned previously, a remote method invocation on an entity bean can be either synchronous or asynchronous. The type of method invocation generally does not have an affect on how
10 the RSM manages state for an invoked entity bean instance.

 In operation 802, the abstract entity bean class order bean 850, which is provided by the application developer, invokes a getter method for container-managed fields and relationships. In response, the concrete implementation class RSM_OrderBean 852, which is generated by the RSM, implements the getter method, in operations 803-805.

15 In operation 806, the abstract entity bean class order bean 850 invokes a setter method for container-managed fields and relationships. In response, the concrete implementation class RSM_OrderBean 852 implements the setter method, in operations 807-809. In this manner, the RSM is able to manage state of container-managed fields and relationships as part of its implementation. In the manage state sequence 800, the
20 RSM_OrderBean 852 uses the OrderState 854 state object to maintain container-managed fields for the OrderBean 850. In addition, the OrderState 854 maintains the relationships of the OrderBean 850 with other entity beans.

5 The RSM 206 uses each SMU instance to track changes to the state objects during a transaction. Each SMU acts as a unit of checkpoint based on its state management type. At transaction commit time, based on the transaction policy, the RSM 206 issues checkpoint for each SMU (and containing State Partition) that participated in the ongoing transaction. The RSM uses the SMU to extract delta of state changes that needs to be checkpointed.

10 Figure 9 is a sequence diagram showing a checkpoint managing sequence 900, in accordance with an embodiment of the present invention. Broadly speaking, when the RSM 206 decides that checkpoint needs to be issued for a set of SMUs, the RSM 206 delegates to a Checkpoint Manager 950 to issue the checkpoints. The Checkpoint Manager 950 extracts the checkpoint state from a SMU and then sends the checkpoint to one of the state servers depending on the type of SMU.

15 More specifically, for each state partition 954 that participated in a particular transaction with a pending checkpoint, the RSM 206 delegates a Checkpoint manager 950 to issue the checkpoint, in operation 901. In response, the checkpoint manager 950 extracts the checkpoint state from the memory replicated SMUs 310 for each SMU of memory replicated type, in operation 902. The Checkpoint manager 950 then sends the checkpoint to the memory replicated state server 214, in operation 903. In a similar manner, the checkpoint manager 950 extracts the checkpoint state from the disk replicated SMUs 308 for each SMU of disk replicated type, in operation 904, and sends the checkpoint to the disk replicated state server 212, in operation 905.

20

Figure 10 is a sequence diagram showing a lock managing sequence 1000, in accordance with an embodiment of the present invention. In operation, the RSM serializes transactions on a state partition by using an exclusive lock per state partition.

In operation 1001, the EJB client 554 invokes a method on an entity bean
5 OrderBean 850. The EJB container 552 intercepts the method invocation and begins a transaction, in operation 1002. In operation 1003, the EJB container 552 then informs the RSM 206 of the transaction begin. The RSM registers a synchronization interface with the container to be informed of transaction calls invoked by the container, in operation 1004. Next, in operation 1005, the EJB container 552 dispatches the method to the
10 RSM_OrderBean 852, which is concrete RSM-specific implementation class for the abstract OrderBean.

The RSM_OrderBean 852 calls the RSM to indicate StatePartitions that will be active as part of this method invocation, in operation 1006. The RSM 206 acquires exclusive locks on all StatePartitions that will be active in this transaction, in operations
15 1007-1008.

Next, in operation 1009, the RSM_OrderBean 852 handles the method invocation using state transition implementation and making setter/getter method invocations on container-managed fields and relationships.

After end of method invocation, the EJB container 552 initiates a commit of the
20 transaction, in operation 1010. The RSM 206 gets a beforeCompletion notification through the registered synchronization interface, in operation 1011. The RSM 206 handles transactionCommitted invocation from transaction manager 1050 (or

LocalTransaction class) by releasing locks on all state partitions that were active in this ongoing transaction, in operations 1012-1014. Now, these state partitions are available for participation in other transactions. After successful lock release, the EJB container 552 sends afterCompletion synchronization notification to the RSM 206, in operation 1015.

5 Generally, the RSM does not propagate any checkpoint until the RSM determines that the on-going transaction will commit successfully. Based on this, RSM has two choices in terms of checkpoint and lock management, namely, delayed lock release and eager lock release.

10 During delayed lock release, the RSM releases the exclusive lock on a state partition after it completes sending the checkpoint to a state server. In the delayed lock release case, the start of the next transaction is delayed because the lock is released only after the successful completion of checkpoint operation for the first transaction.

15 In eager lock release, the RSM releases the exclusive lock on a state partition as part of the commit operation for the first transaction. The lock is release after the checkpoint state has been extracted from the participating SMUs, but before a checkpoint operation has been issued. After the exclusive lock is released for the first transaction, the RSM allows a blocked second transaction to acquire the exclusive lock and operate on the state partition. In this case, the start of the second transaction is not delayed by the completion of the checkpoint operation for the first transaction.

20 Although the foregoing invention has been described in some detail for purposes of clarity of understanding, it will be apparent that certain changes and modifications may be practiced within the scope of the appended claims. Accordingly, the present

embodiments are to be considered as illustrative and not restrictive, and the invention is not to be limited to the details given herein, but may be modified within the scope and equivalents of the appended claims.

What is claimed is:

FOUO: 674430